Лабораторна робота №2

Знайомство із середовищем розробки програм Microsoft Visual Studio

Мета: Отримати перші навички роботи з Microsoft Visual Studio для створення програм, написаних мовою асемблера, а також вивчити команди MOV та CPUID.

Завдання:

1. Створити у середовищі MS Visual Studio проект з ім'ям **Lab2**. Встановити необхідні параметри проекту – опції середовища розробки програм.
2. Написати вихідний текст програми на асемблері, додати файл вихідного тексту у проект. Зміст вихідного тексту згідно з варіантом завдання.
3. Скомпілювати вихідний текст і отримати виконуємий файл програми.
4. Перевірити роботу програми. Налагодити програму.
5. Отримати дизасембльований текст машинного коду і проаналізувати його.

Теоретичні відомості

Асемблер і Microsoft Visual Studio

Система Microsoft Visual Studio призначена для розробки програм на різноманітних мовах програмування – C#, Visual Basic, C++, а також асемблері.

Необхідно відзначити, що оскільки у середовищі MS Visual Studio для компіляції асемблерних файлів використовується компілятор MASM (подібний до MASM32), то синтаксис вихідних текстів подібний до розглянутого вище у лабораторній роботі №1. Таким чином, теоретичні положення щодо мови асемблеру, викладені у розділі теоретичних відомостей для попередньої лабораторної роботи №1, є чинними і для роботи №2.

Ця робота присвячена розробці програм типу Win32 – 32-бітних програм, які працюють у відповідному середовищі операційної системи Windows. Скелет вихідного тексту на асемблері для таких програм повністю повторює вже розглянутий вище у попередній роботі. Більше того, як передбачається, можуть бути використані деякі файли зі складу пакету MASM32 – а саме бібліотек функцій API Windows.

Команда MOV

Ця команда часто використовується у програмах на асемблері. Вона виконує копіювання даних. Команда MOV має два операнди:

```
mov Куди, Джерело
```

Операнд **Джерело** повинен вказувати, звідки взяти інформацію. Перший операнд вказує, куди записати інформацію. Наприклад:

```
mov ecx, eax
```

означає скопіювати дані з регістру EAX у регістр ECX. У наступному рядку

```
mov ax, 5
```

запрограмований запис числа 5 у регістр AX. У якості другого операнду записане безпосередньо числове значення. Такі значення зберігаються у пам’яті, тому фактично виконується копіювання типу "пам’ять → регістр".

Певна кількість байтів джерела повинна записуватися у відповідне за розміром місце. Приклади помилок:

```
mov al, edx       ;помилка: з 32-бітового у 8-бітовий регістр
mov ax, ecx       ;помилка: з 32-бітового у 16-бітовий регістр
mov eax, cx       ;помилка: з 16-бітового у 32-бітовий регістр
```

Наступні приклади:

```
mov al, 5         ; AL = 00000101 (8 біт)
mov ax, 5         ; AX = 0000000000000101 (16 біт)
mov eax, 5        ;EAX = 00000000000000000000000000000101 (32 біти)
```

Тут помилок немає – у регістри записується відповідна кількість бітів значення, яке може представлятися як 8-бітовим, так і 16-, або 32-бітовим двійковим кодом. У той же час запис

```
mov al, 259    ;помилка: у 8-бітовий регістр число 259 не можна
```

неприпустимий – для представлення числа 259 восьми бітів замало. Асемблер при компіляції видасть помилку.

Можна сказати, що форма запису на асемблері

```
mov a, b
```

означає операцію присвоювання у мові програмування високого рівня:

a = b

Можна розглянути присвоєння значення однієї перемінної іншій, запрограмоване мовою C/C++:

```
long a, b;

a = b;
```

Записати відповідний код на асемблері можна спробувати так:

```
.data
  a dd ?            ; створення неініціалізованої перемінної a
  b dd ?            ; створення неініціалізованої перемінної b

.code

mov a, b            ; помилка, так не можна
```

Перемінні a та b є об'єктами у пам'яті. Одною командою MOV копіювати з пам'яті у пам'ять не можна. Можна, наприклад, так:

```
mov eax, b          ; регістр EAX у якості посередника
mov a, eax
```

Необхідно зазначити, що у наведених вище рядках використання імен перемінних у якості операндів команд MOV дещо спрощує синтаксис, проте ховає те, що насправді замість імен використовуються адреси відповідних перемінних. Більш адекватний програмний код того, що насправді виконується, можна отримати у вікні дизасемблера.

```
mov eax, dword ptr [b]
mov dword ptr [a], eax
```

У мові асемблера квадратні дужки означають, що всередині них міститься вказівник – той, хто зберігає адресу пам'яті.

Синтаксис мови асемблера MASM достатньо гнучкий. Запис

```
mov eax, b
```

можна вважати дещо спрощеним варіантом, аніж

```
mov eax, [b]
```

який, у свою чергу, є спрощенням від:

```
mov eax, dword ptr [b]
```

Остання форма запису є найбільш коректною.

Розглянемо наступний приклад. Створимо масив з чотирьох 32-бітових елементів – елементів типу DWORD:

```
.data
  M dd 4 dup(?)
```

Елементи масиву розташовуються у пам’яті послідовно. Загалом для масиву потрібно 16 байтів пам’яті (рис. 1)

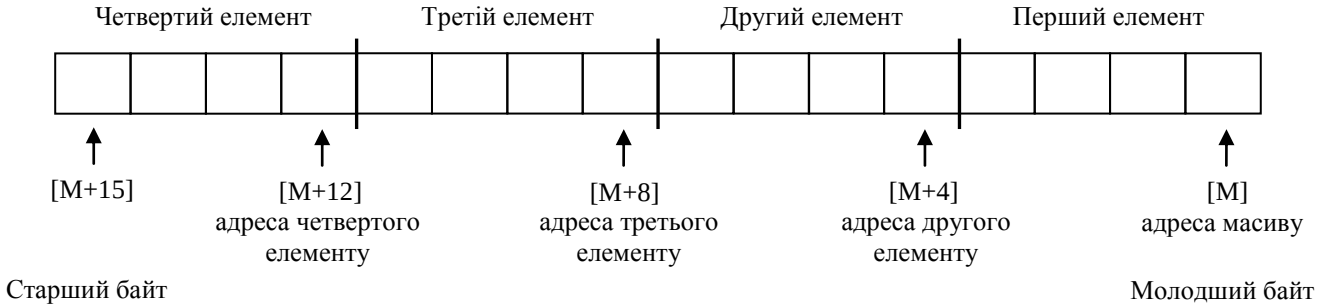


Рис. 1. Масив з чотирьох елементів типу DWORD

Запишемо у другий елемент якесь значення, наприклад:

```
mov dword ptr[M+4], 89ABCDEFh
```

Результат на рис. 2.

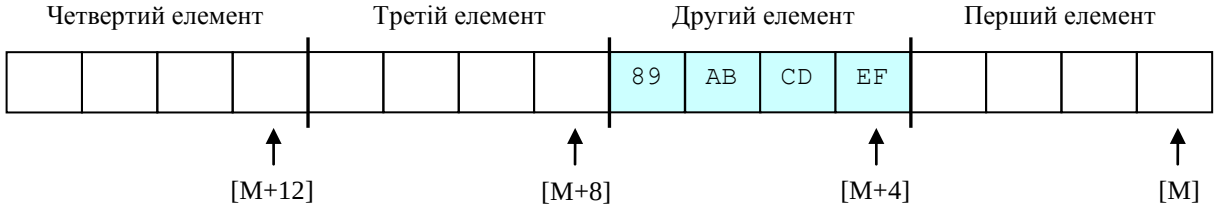


Рис. 2

Ще приклад:

```
mov dword ptr[M+7], 10CCABBAh
```

Результат на рис. 3.

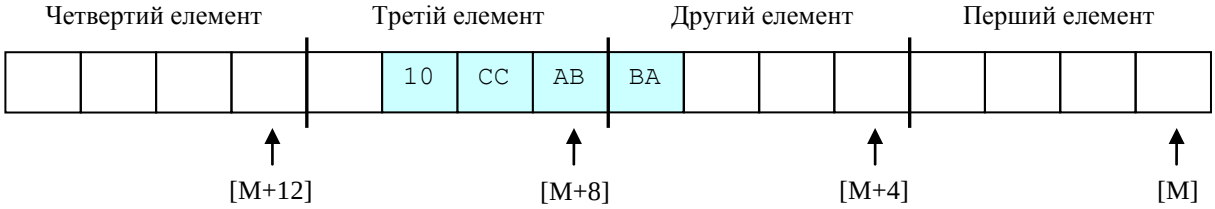


Рис. 3

Приклад запису двохбайтового значення у пам’ять

```
mov word ptr[M+8], 4352h
```

Результат на рис. 4.

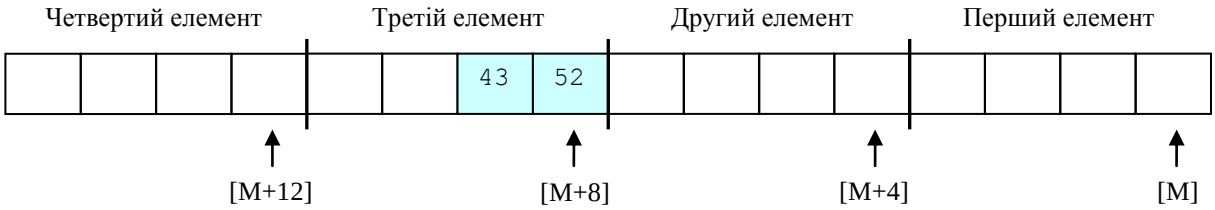


Рис. 4. Запис двох байтів у пам’ять

Приклад запису однобайтового значення у пам’ять

```
mov byte ptr[M+9], 2Ah
```

Результат на рис. 5.

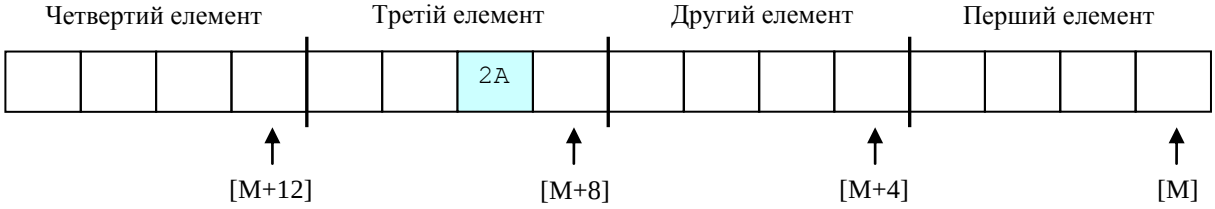


Рис. 5. Запис одного байту у пам’ять

Команда CPUID

Ця команда призначена для ідентифікації процесора та отримання відомостей про його властивості. Для того, щоб вказати, які саме відомості потрібні, треба записати у регістр EAX деяке значення – параметр для команди. Результати роботи команди CPUID процесор записує у регістри EAX, EBX, ECX EDX.

Для того, щоб правильно користуватися командою CPUID, потрібно дотримуватися певної послідовності роботи програми. Можна спочатку перевірити – а чи можливо взагалі користуватися цією командою? Для цього треба перевірити значення 21-го біту регістру EFLAGS. Проте, у цій лабораторній роботі можна вважати це зайвим.

Отримання відомостей за допомогою команди CPUID робиться у декілька кроків. Спочатку треба виконати цю команду із значенням параметру 0:

```
mov eax,0
cpuid
```

Після виконання такого коду, у регістрі EAX міститься значення, яке означає максимально можливе значення параметру, яке можна використати для отримання базових відомостей. Якщо намагатися викликати команду CPUID із параметром більше зазначеного вище максимального, то результатом будуть усі нулі (це не стосується параметрів так званих розширених відомостей).

Після виконання команди CPUID із параметром 0 процесор також записує у регістри EBX, ECX та EDX коди символів імені процесора. Для процесорів Intel це буде "GenuineIntel", для процесорів AMD – "AuthenticAMD" тощо. Необхідно відзначити, що 12 символів записуються четвірками у такому порядку – спочатку EBX, потім EDX, останні чотири у регістрі ECX.

Наступним кроком у програмі буде виконання команди з параметром 1:

```
mov eax,1
cpuid
```

Після виконання цієї команди у регістри EAX, EBX, ECX та EDX буде записано інформацію про сімейство процесорів, модель та деякі інші відомості.

Наступним кроком у програмі буде виконання команди з параметром 2:

```
mov eax,2
cpuid
```

і так далі, наскільки можливо. Як вже вказувалося вище, максимально можливе значення параметру стає відомим після виконання команди з параметром 0. Це для базового набору відомостей. Проте, є ще так звані, розширені відомості, які відповідають функціям на основі команди CPUID з параметром 80000000h і більше. Для того, щоб дізнатися максимальне значення для параметру розширених функцій, потрібно виконати:

```
mov eax,80000000h
cpuid
```

У результаті виконання цього у регістрі EAX буде деяке значення, наприклад, 80000008h. Можна послідовно виконувати команди CPUID із значеннями параметрів від 80000001h до 80000008h для отримання відповідних відомостей. Наприклад, після виконання коду:

```
mov eax,80000008h
cpuid
```

у регістрі EAX буде інформація про максимальну можливу розрядність адрес пам’яті даного процесора.

Вичерпна інформація щодо CPUID міститься у документі "Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference", який можна завантажити через інтернет по адресі http://www.intel.com.

У підсумку, виконання ланцюжка команд CPUID можна відобразити так:

```
;---базові функції---
mov eax,0     ;початок
cpuid
. . .         ;зберігання значень EAX,EBX,ECX,EDX
mov eax,1
cpuid
. . .         ;зберігання значень EAX,EBX,ECX,EDX

. . .         ;якщо можливо, то інші функції для EAX>1

;---розширені  функції---
mov eax,80000000h
cpuid
. . .         ;зберігання значень EAX,EBX,ECX,EDX
mov eax,80000001h   ;якщо можливо
cpuid
. . .         ;зберігання значень EAX,EBX,ECX,EDX

. . .         ;якщо можливо, то інші функції для EAX>80000001h
```

Для зберігання значень регістрів EAX, EBX, ECX та EDX можна їх записувати у масив-вектор, наприклад:

```
.data
  res dd 256 dup(0)

.code
  mov eax, 0
  cpuid
  mov dword ptr[res], eax
  mov dword ptr[res+4], ebx
  mov dword ptr[res+8], ecx
  mov dword ptr[res+12], edx
  . . .         ;у подібний спосіб і для інших CPUID
```

Проте, зберігання четвірок 32-бітових значень можна запрограмувати, як здається, і по-іншому.

Вивід інформації

Текстова інформація може відображатися достатньо простими засобами – викликом стандартних діалогових вікон MessageBox. Для цього достатньо сформувати у двох буферах потрібний текст: основний і текст заголовку:

```
.data
  Text db 256 dup(0)
  Caption db 32 dup(0)

.code
  . . .      ;формування вмісту буферів Text, Caption

  invoke MessageBoxA, 0, ADDR Text, ADDR Caption, 0
```

Як сформувати вміст буферів тексту? Для даної роботи потрібно виводити значення, отримані командами CPUID. Числові значення можна показувати у шістнадцятковому коді. Для цього потрібно перетворювати 32-бітові значення типу DWORD у 8 символів шістнадцяткових цифр. Таке перетворення можна виконати за допомогою окремої процедури **DwordToStrHex**. Процедура викликається наступним чином:

```
  push ...                ;числове 32-бітове значення
  push offset ...         ;адреса буферу тексту
  call DwordToStrHex
```

Вихідний текст процедури **DwordToStrHex** пропонується нижче:

```
;ця процедура записує 8 символів HEX коду числа
;перший параметр - 32-бітове число
;другий параметр - адреса буфера тексту
DwordToStrHex proc
   push ebp
   mov ebp,esp
   mov ebx,[ebp+8]     ;другий параметр
   mov edx,[ebp+12]    ;перший параметр
   xor eax,eax
   mov edi,7
@next:
   mov al,dl
   and al,0Fh          ;виділяємо одну шістнадцяткову цифру
   add ax,48           ;так можна тільки для цифр 0-9
   cmp ax,58
   jl @store
   add ax,7            ;для цифр A,B,C,D,E,F
@store:
   mov [ebx+edi],al
   shr edx,4
   dec edi
   cmp edi,0
   jge @next
   pop ebp
   ret 8
DwordToStrHex endp
```

Формування символів шістнадцяткових кодів для значень EAX, EBX, ECX, EDX, збережених у масиві **res**, може бути запрограмоване, наприклад, так:

```
.data
  res dd 256 dup(0)
  Text db 'EAX=xxxxxxxx',13,10,
          'EBX=xxxxxxxx',13,10,
          'ECX=xxxxxxxx',13,10,
          'EDX=xxxxxxxx',0
  Caption db "Результат CPUID 0",0

.code
  . . .                   ;інший програмний код (заповнення масиву res)

  push [res]              ;значення регістру EAX з масиву res
  push offset [Text+4]    ;адреса, куди записуються 8 символів
  call DwordToStrHex
  push [res+4]            ;значення регістру EBX з масиву res
  push offset [Text+18]
  call DwordToStrHex
  push [res+8]            ;значення регістру ECX з масиву res
  push offset [Text+32]
  call DwordToStrHex
  push [res+12]           ;значення регістру EDX з масиву res
  push offset [Text+46]
  call DwordToStrHex
  invoke MessageBoxA, 0, ADDR Text, ADDR Caption, 0
```

Таким чином, можна отримати, наприклад, такий текст (рис. 1):

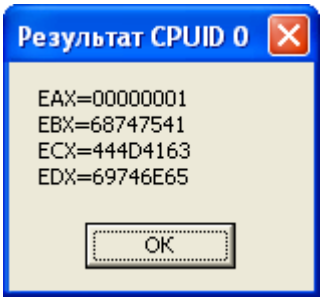


Рис. 1. Відображення значень після виконання команди CPUID

Як окремий випадок можна відзначити вивід імені процесора, яке записується у регістри EBX, ECX та EDX після виконання команди з параметром CPUID 0. Особливість полягає у тому, що у вказані регістри тут записуються безпосередньо однобайтові ASCII коди символів – по чотири символи в регістр. Тоді ці коди символів можна прямо вставити у відповідні байти текстового буферу. Наприклад:

```
.data
  Vendor db 16 dup(0)
  CaptionVendor db "CPUID 0 Vendor string",0
  . . .                    ;інші дані

.code
  mov eax, 0
  cpuid
  mov dword ptr[Vendor], ebx
  mov dword ptr[Vendor+4], edx
  mov dword ptr[Vendor+8], ecx
  . . .                    ;інший програмний код

  invoke MessageBoxA, 0, ADDR Vendor, ADDR CaptionVendor, 0
```

У діалоговому вікні буде відображатися наступне (рис. 2):

Рис. 2. Один з результатів виконання команди CPUID

Стосовно конфіденційності. Команда CPUID дозволяє встановити приналежність процесора тільки до певного типу. Індивідуально процесори не ідентифікуються, хоча колись така можливість розглядалася, проте потім, як здається, розробники процесорів архітектури x86 від цього відмовилися.

Порядок виконання роботи та методичні рекомендації

При вивченні системи Microsoft Visual Studio треба звернути увагу на те, що є різні версії цієї системи відповідно різним рокам випуску та мовам інтерфейсу. Україномовної версії поки що немає. Опис порядку виконання роботи та методичні рекомендації тут конкретизовані для англомовної версії MS Visual Studio 2013.

1. Створення проекту у середовищі MS Visual Studio

При створенні проекту чисто на асемблері потрібно враховувати особливості середовища розробки програм. Першою особливістю є те, що для проектів MS Visual Studio у переліку мов програмування асемблер відсутній. Рекомендація: вкажіть у списку мов проект Visual C++.

Виберіть меню "File – New – Project", і далі у діалоговому вікні "New project" виберіть Visual C++, а потім – Win32 project (рис.3):

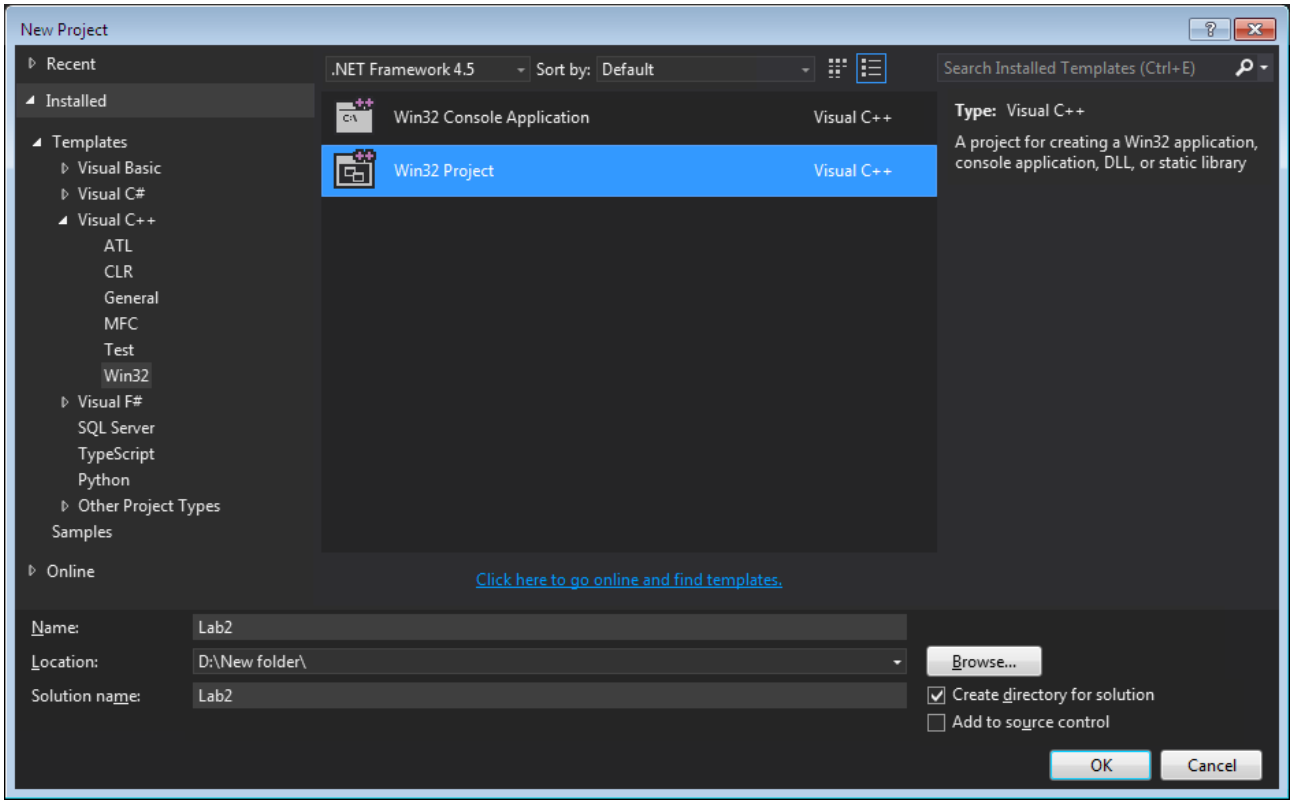


Рис. 3.

Потрібно ввести ім'я програми. Крім того, тут можна вказати робочу папку проекту – за умовчанням це New folder. Нехай ім'я програми та ім'я рішення буде, наприклад, Lab2. Вводимо і натискуємо кнопку ОК.

З'являється вікно "Win 32 Application wizard". Натискуємо кнопку "Next >". З'являється наступна сторінка (рис. 4)

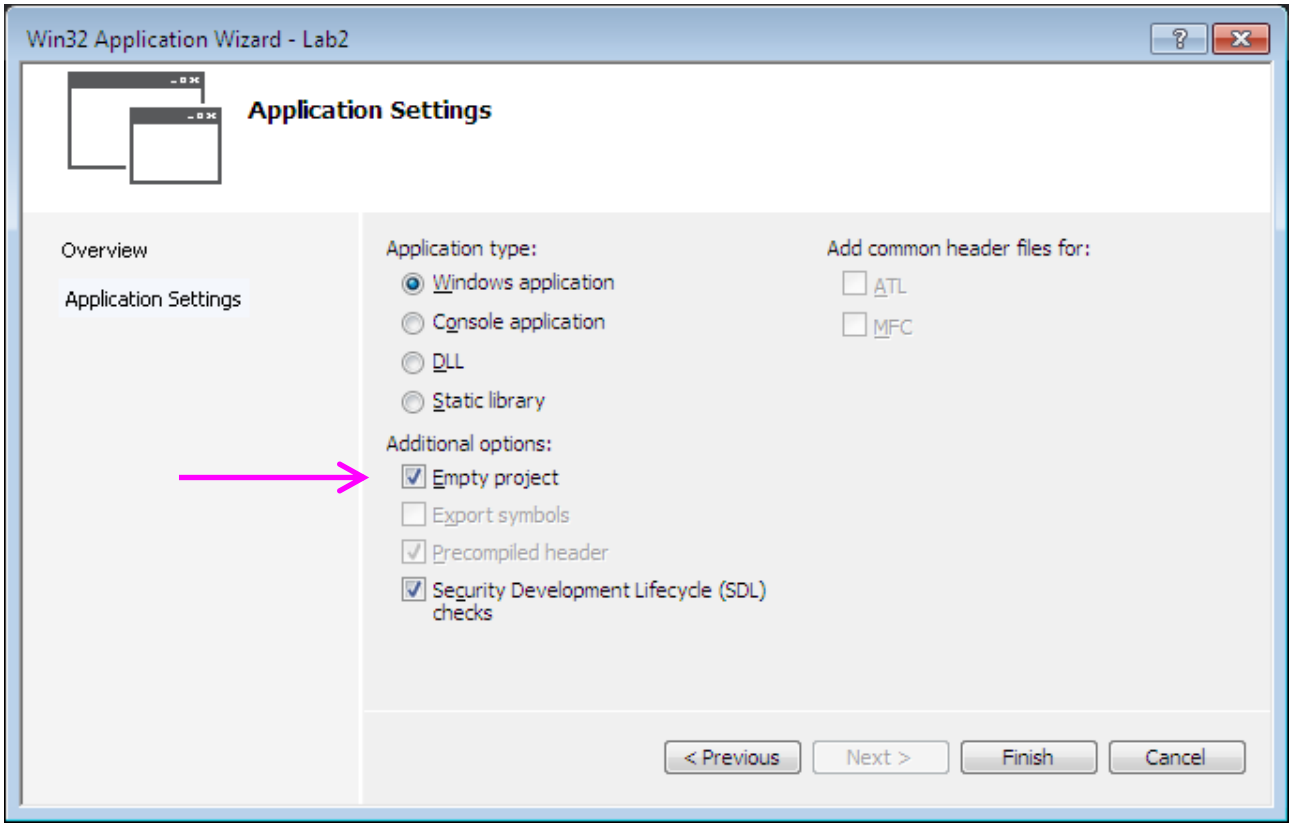


Рис. 4. Основні параметри проекту програми Lab2

Вибираємо "Empty Project", поставивши відповідну позначку у групі "Additional options". Натискуємо кнопку "Finish". Середовище Visual Studio переходить у стан роботи з новоствореним проектом. У вікні "Solution Explorer" показується такий список (рис. 5)

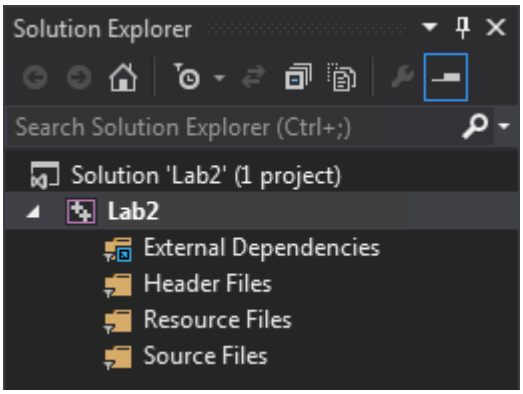


Рис. 5.

Оскільки ми створили порожній проект, то жодних вихідних текстів у вказаних розділах немає. Потрібно додавати тексти програми у папку **Source Files** та у інші папки (якщо потрібно). Проте, спочатку необхідно дещо налагодити

робоче середовище Visual Studio, інакше асемблерні тексти не будуть сприйматися.

2. Налагодження підтримки мови асемблера

У вікні " Solution Explorer " встановіть курсор на імені проекту (Lab2) і клацніть правою кнопкою миші. У спливаючому меню виберіть пункт "Build Dependencies" і далі "Build Customization" (рис. 6).

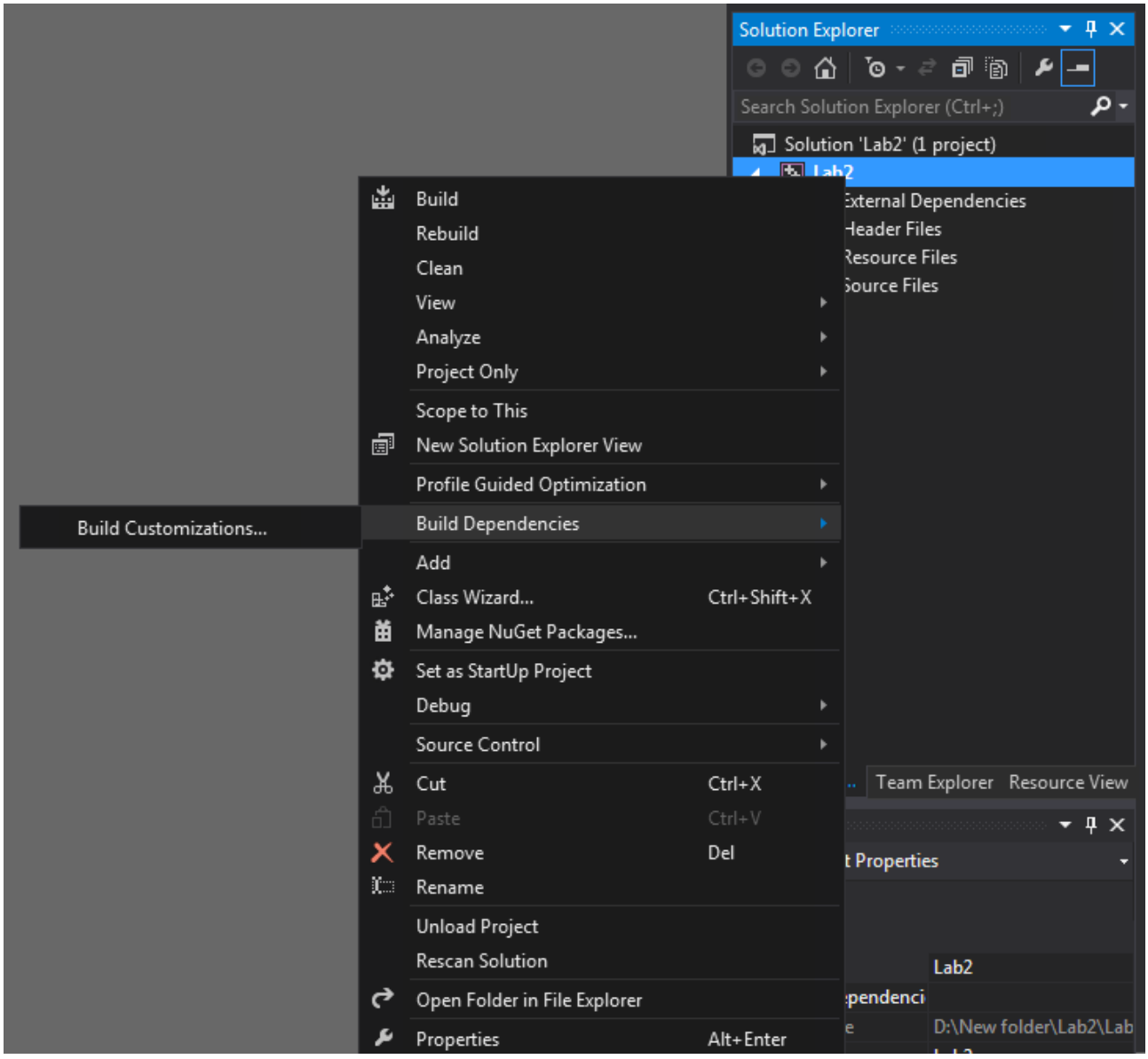


Рис. 6.

Далі з'явиться відповідне діалогове вікно (рис. 7)

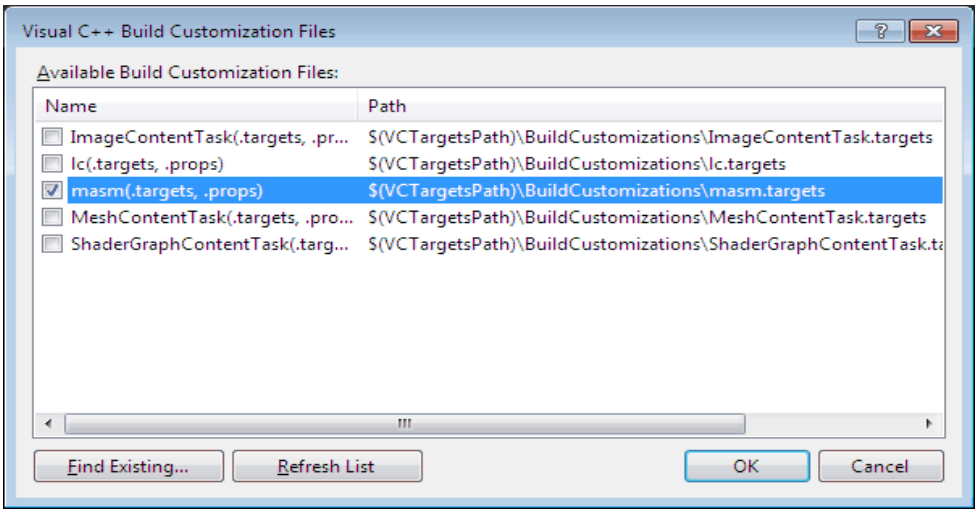


Рис. 7.

Виберіть пункт "masm", та зробіть позначку у квадратику. Натисніть ОК.
Таким чином, у проекті активується вбудований у Visual Studio асемблер MASM, який буде компілювати файли вихідних текстів *.asm.

3. Додавання вихідних текстів у проект

Нам потрібно додати у проект файл вихідного тексту на асемблері. Будемо вважати, що тексту поки що немає і ми плануємо його писати безпосередньо у середовищі Visual Studio. Вкажіть курсором розділ "Source Files", клацніть правою кнопкою миші, і потім - "Add" – "New Item…" (рис. 8)

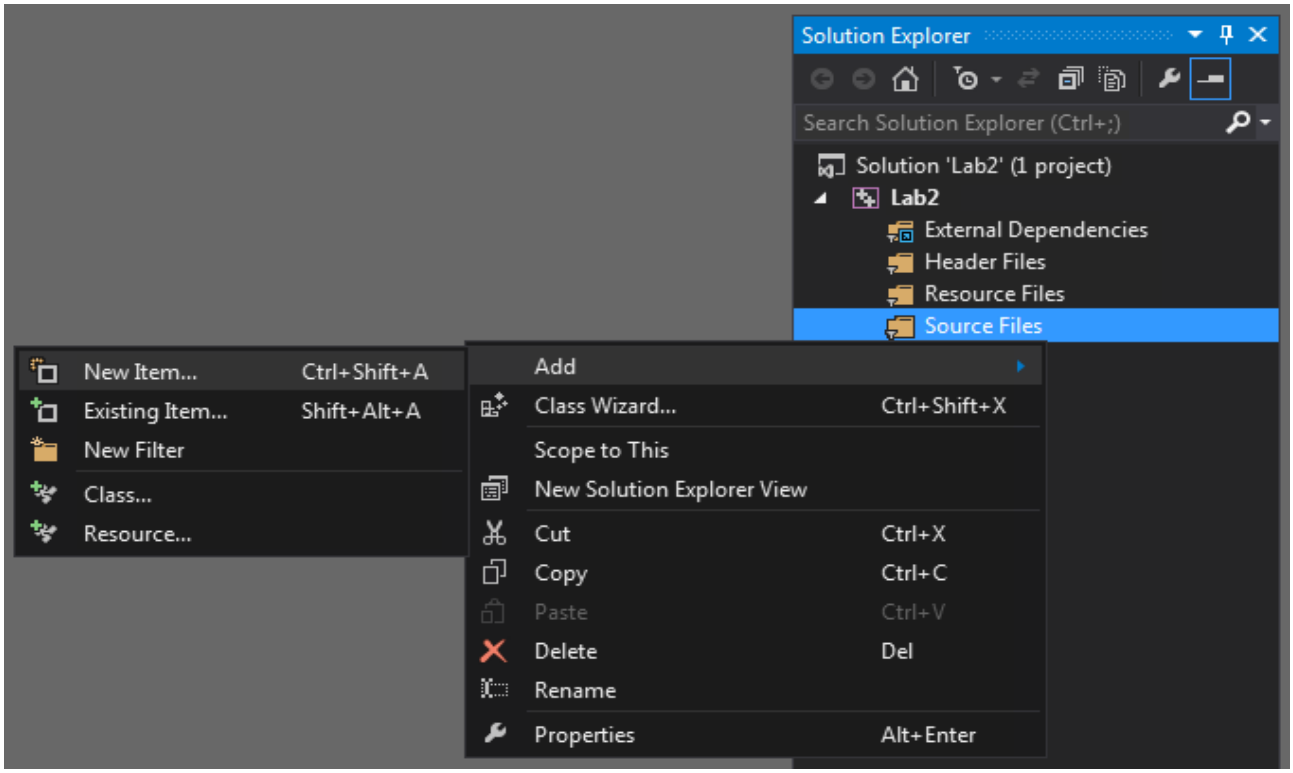
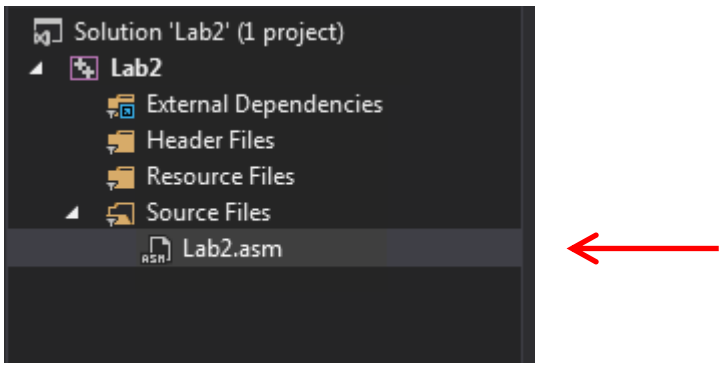


Рис. 8.

З'являється відповідне вікно, у якому буде запропоновано вибрати тип файлу та його назву. Нам потрібен файл вихідного тексту Асемблера – проте у цьому списку таких типів файлів немає. Можемо вибрати майже будь-який тип, наприклад, тип файлів .cpp, але ввести ім'я та розширення – **Lab2.asm** і натиснути кнопку додавання. У вікні Solution Explorer у розділі Source Files повинен з'явитися Lab2.asm



Також у середовищі Visual Studio відкривається нове порожнє вікно з ім'ям lab2.asm, у якому ми будемо писати текст програми. Наприклад, такий:

```
.586
.model flat, stdcall

include \masm32\include\kernel32.inc
include \masm32\include\user32.inc

includelib \lib\kernel32.lib
includelib \lib\user32.lib

.data
  Caption db "Я програма на асемблері",0
  Text db "Здоровеньки були!",0

.code
main:
  invoke MessageBoxA, 0, ADDR Text, ADDR Caption, 0
  invoke ExitProcess, 0
end main
```

Ця програма має виводити текст-вітання "Здоровеньки були!" у діалоговому вікні із заголовком "Я програма на асемблері".

4. Підключення бібліотек

Програма викликає дві функції Windows, які доступні через інтерфейс API Win32:

- **MessageBoxA** для показу простого діалогового вікна. Ця функція міститься у системній DLL – **user32.dll**. Для побудови проекту лінкер використовує бібліотеку імпорту **user32.lib**.

- **ExitProcess** – цю функцію повинна викликати будь-яка програма для коректного завершення. Вона викликається з системної DLL – **kernel32.dll**. Для побудови проекту використовується бібліотека імпорту **kernel32.lib**.

Підключення вказаних бібліотек до проекту описується рядками

```
includelib \lib\kernel32.lib
includelib \lib\user32.lib
```

Для компіляції викликів функцій **MessageBoxA** та **ExitProcess** потрібно знати, які вони мають аргументи. Опис аргументів міститься у заголовочних файлах ***.inc**, які включені у текст директивами include:

```
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
```

Де взяти файли **kernel32.inc** та **user32.inc**? Серед файлів Visual Studio їх немає. Вказані файли можна взяти з комплекту MASM32 (дивіться лабораторну роботу №1). Потрібно створити на диску, на якому записується поточний проект, наприклад, диску D, папку **d:\masm32\include** і записати туди вказані вище файли.

5. Компіляція та створення виконуємого файлу програми

Для створення виконуємого файлу програми потрібно спочатку скомпілювати вихідний текст, а потім лінкер запише файл – lab2.exe. Потім викликати програму на виконання. Усе це можна зробити одразу через меню "Debug – Run".

У разі успішної компіляції, лінкування, запису файлу **lab2.exe** і виклику програми, ця програма запрацює і покаже вікно-вітання (рис. 11):

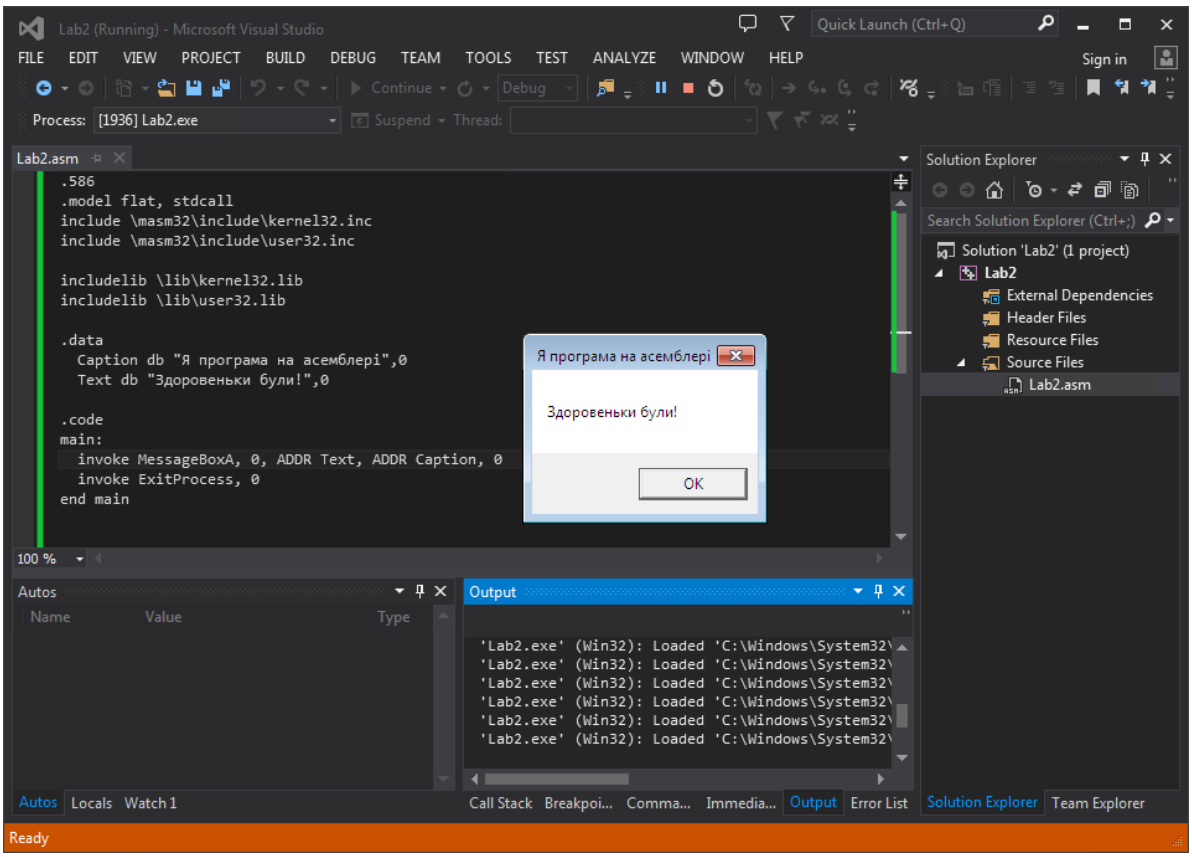


Рис. 11. Вигляд вікна програми Lab2 на тлі середовища MS Visual Studio

Натисніть кнопку "OK" – програма закінчить свою роботу.

6. Debug- та Release-версії програми

За умовчанням при створенні нового проекту середовище Visual Studio відпрацьовує Debug-версію нашої програми. Що це означає? У машинний код вставляються команди, які дозволяють контролювати роботу програми, налагоджувати її. У робочій папці створюється вкладена папка Debug – у нашому проекті це \Lab2\Debug у яку після кожного успішного сеансу лінкування записується файл lab2.exe. Debug-версія програми є деяким тимчасовим зразком, який має сенс тільки на етапі розробки та налагодження.

Якщо програміст хоче розповсюджувати власну програму, то Debug-версія для цього не дуже придатна з багатьох причин.

Фінальна версія exe-файлу програми, яка призначена для розповсюдження і використання на інших комп'ютерах, зветься Release-версією. Щоб налаштувати конфігурацію середовища Visual Studio на створення цієї версії, виберіть меню Build – Configuration. Далі у наступному вікні виберіть "Release" (рис. 12).

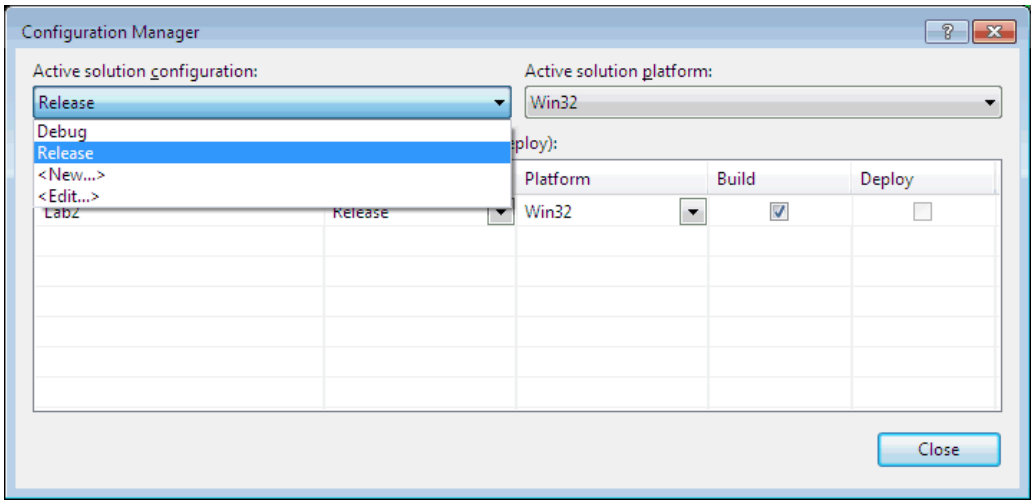


Рис. 12. Зміна конфігурації Debug на Release

Середовище починає працювати трохи по-іншому. Набір властивостей компілятора, лінкера тощо тут зветься конфігурацією. Виконуємий файл буде записуватися у вкладену папку Release - у нашому проекті це буде \Lab2\Release. Порівняйте файли lab2.exe у папці Release та у папці Debug.

Необхідно враховувати відмінність багатьох параметрів роботи середовища для різних конфігурацій. Це стосується також і параметрів (опцій) роботи компілятора та лінкера. Може так трапитися, що компілятор або лінкер можуть видати повідомлення про помилку, якої не було у Debug-версії (рис 13)

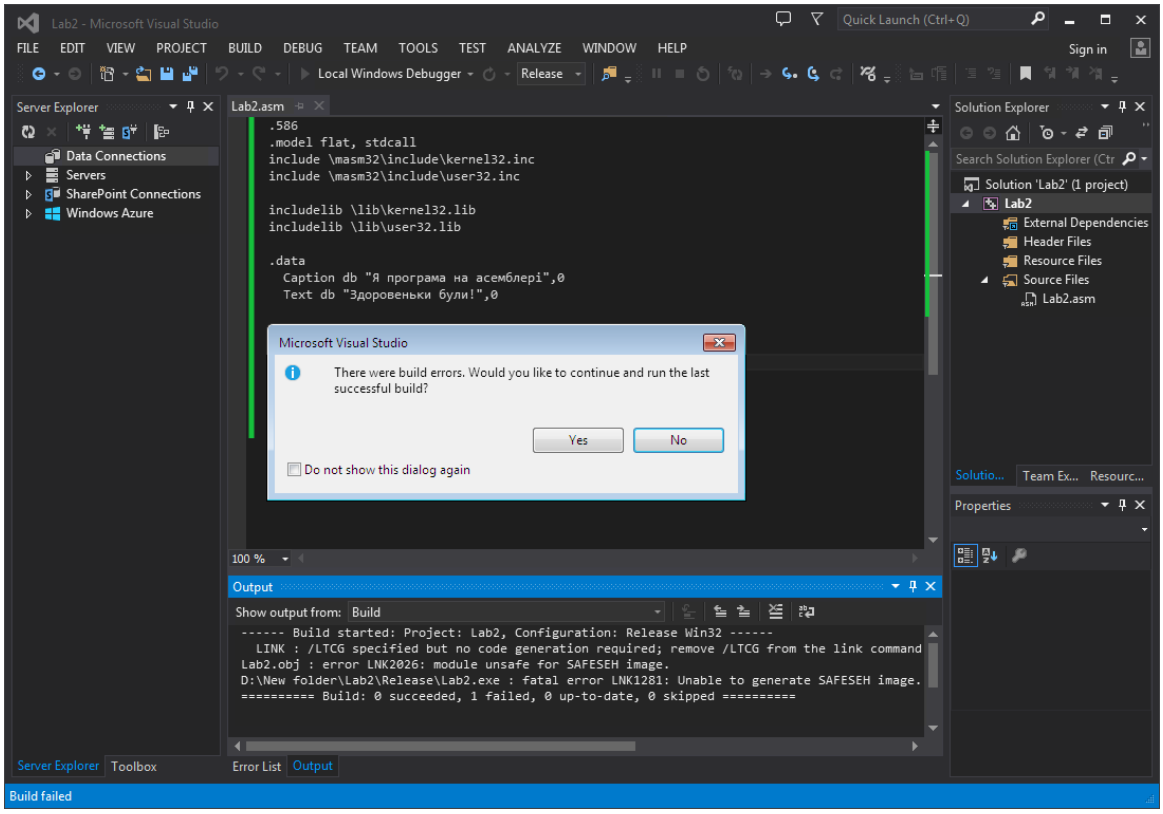


Рис. 13. Лінкер зупинив побудову і повідомив про помилку

Зазвичай, компілятор та лінкер надають деякі відомості про помилки, зокрема, можна придивитися до вмісту вікна "Output" (рис. 14)

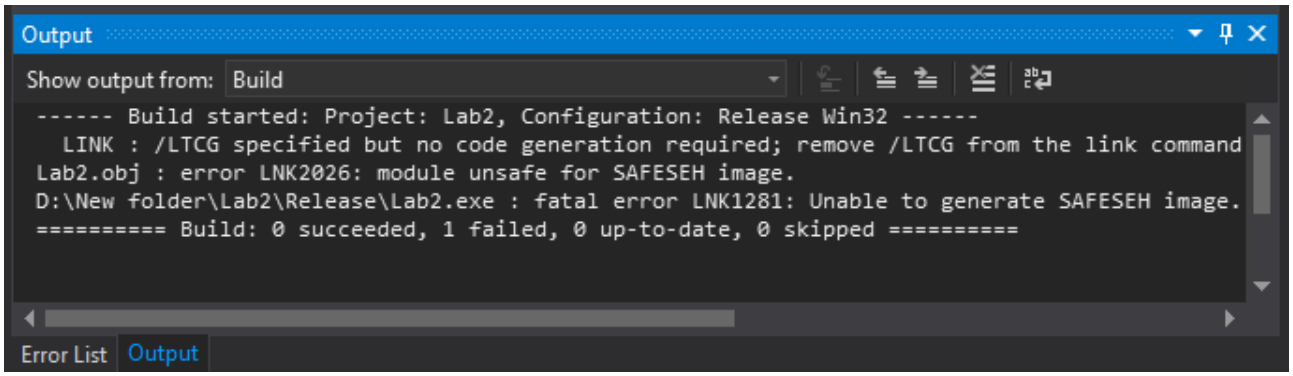


Рис. 14. Збільшений вигляд нижньої частини попереднього Рис. 13

Тут мова йде про якісь /LTCG та SAFESEH

У будь-якому випадку помилок потрібно у вікні повідомлень натиснути кнопку "No" і шукати вирішення проблеми. Середовище призупиняє роботу (рис. 15).

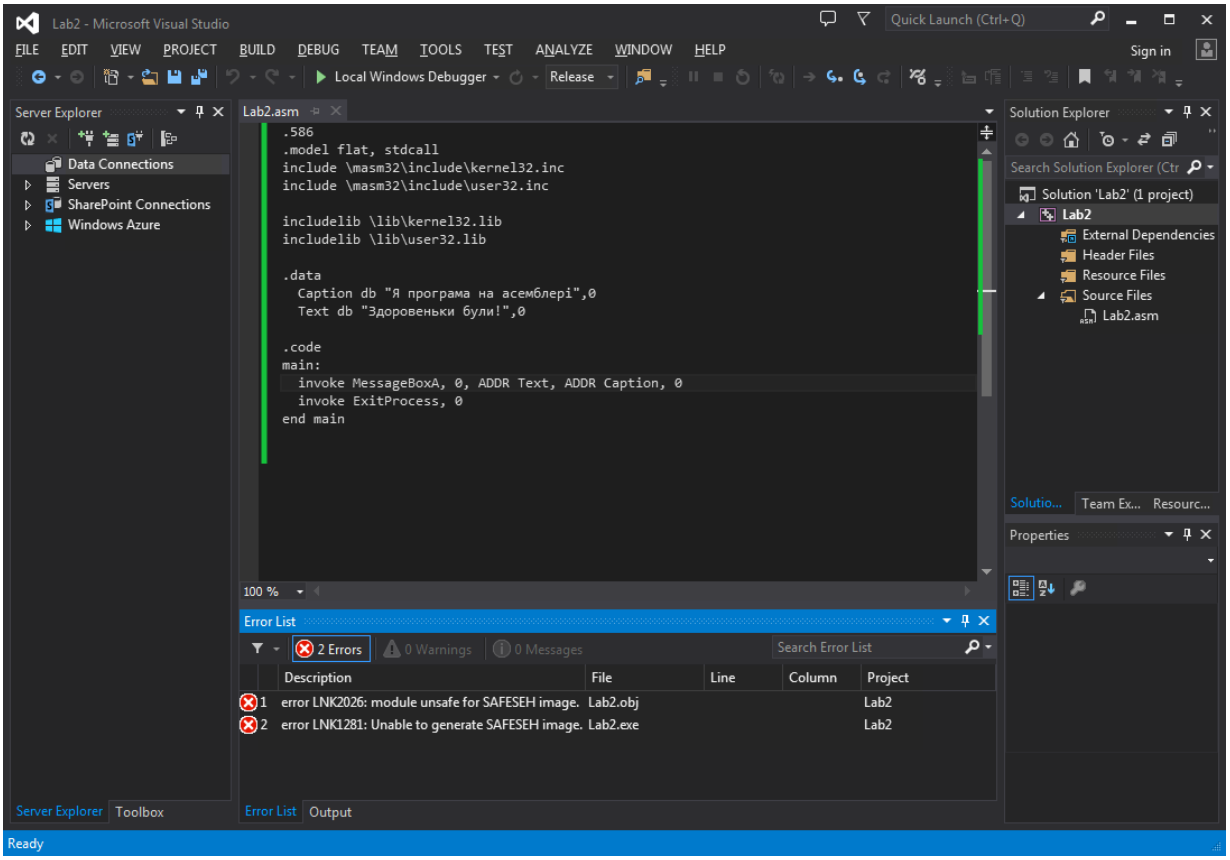


Рис. 15. Список помилок у вікні "Error List" знизу

Де шукати якійсь SAVESEH? Оскільки таку помилку видає лінкер, то можна спробувати пошукати у налаштуваннях лінкера.

Вибираємо пункт меню Project – Properties і у вікні властивостей проекту розкриваємо розділ Linker. На сторінці Advanced міститься рядок із SAFESEH. Вказуємо No замість Yes (Рис. 16)

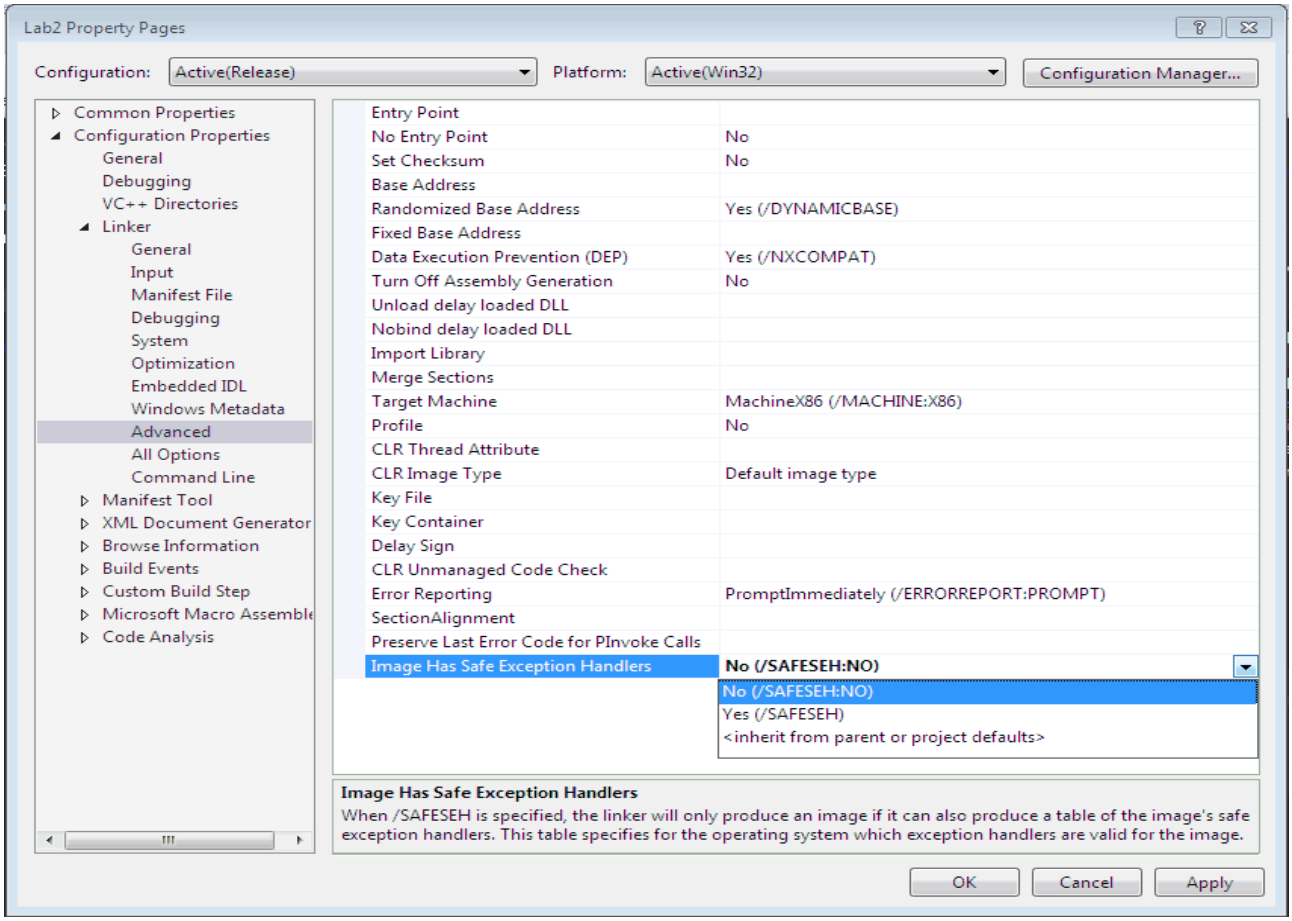


Рис. 16. Одна зі сторінок опцій налаштування лінкера MS Visual Studio

Після внесення змін, натискуємо кнопку Apply або Ok.

Необхідно зазначити, що середовище MS Visual Studio є достатньо складним, і не завжди можна швидко розібратися в усіх його опціях та налаштуваннях, у тому числі, при переході на іншу версію цього середовища розробки програм.

Можна вказати ще на один аспект при використанні мови програмування Асемблер у середовищі Visual Studio – точка входу у програму. У тексті нашої програми вона позначена міткою "**main:**". Зазвичай Visual Studio успішно автоматично знаходить точку входу в програму, навіть якщо вона позначається

не "**main:**" а будь-якою іншою міткою, наприклад, "**start:**" або взагалі довільним набором англомовних символів. Проте, інколи можуть виникати помилки, повязані з тим, що точка входу автоматично не знаходиться. Тоді її треба для Visual Studio вказати явно. Для цього через меню меню Project – Properties у вікні властивостей проекту розкриваємо розділ Linker. У першому рядку знаходимо Entry Point і вписуємо туди потрібну назву – **main**, або те що треба (рис. 17)

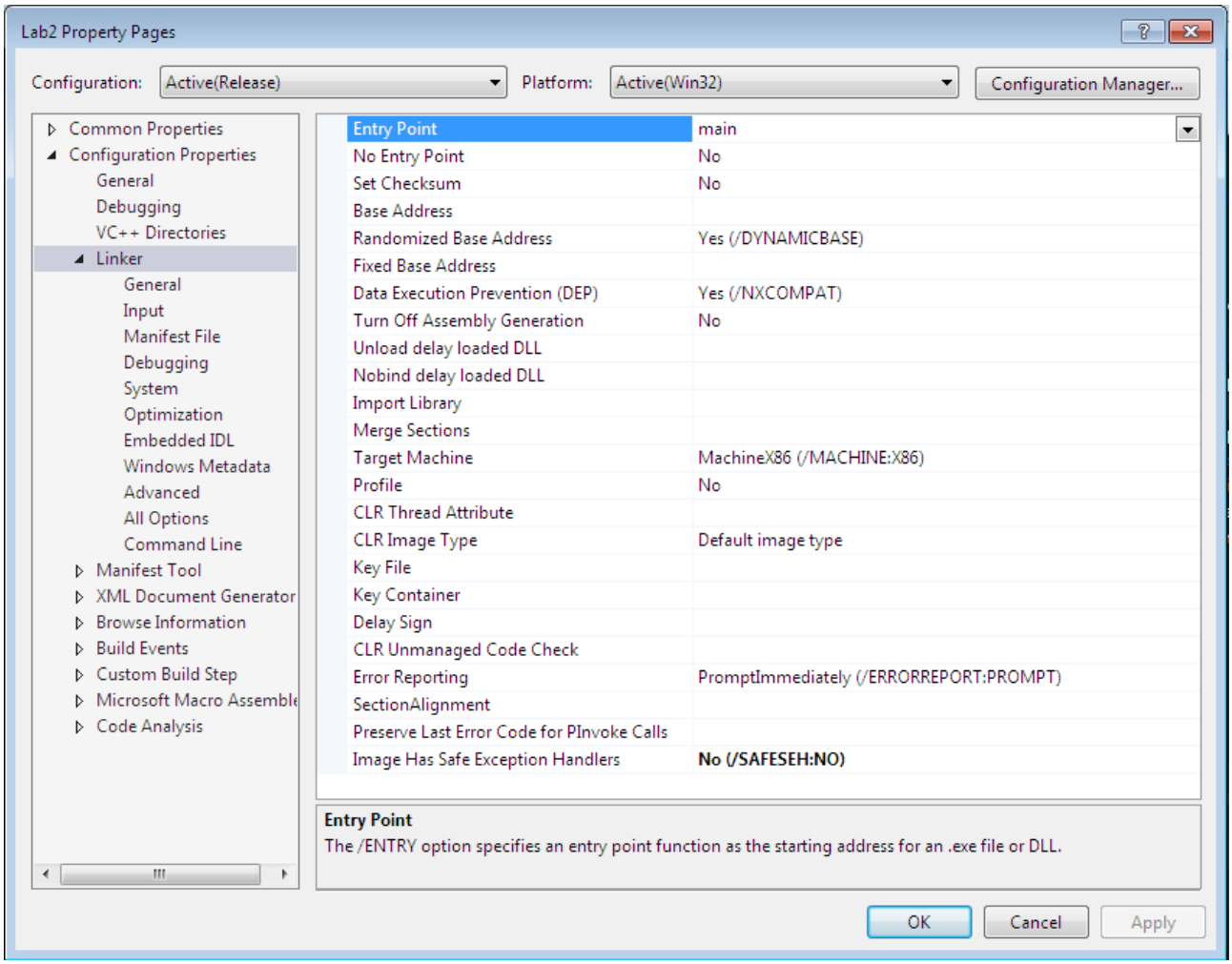


Рис. 17. Вказування імені точки входу

7. Налагодження програми, огляд та аналіз машинного коду

Зазвичай програму потрібно налагоджувати, виправляти помилки, вдосконалювати. Після кожного кроку виправлення помилок можна робити перевірку – спробувати побудувати проект. Це робиться через меню Build. Проте, зазвичай зручніше усі дії виконувати через меню Debug – Run (кнопка F5 на клавіатурі) – автоматично виконуються усі етапи: компіляція, лінкування і при відсутності помилок програма буде викликана для демонстрації роботи або відстеження ходу її виконання.

В принципі, налагоджувати программу можна як у Debug – конфігурації, так і у Release. Набагато більше можливостей у конфігурації Debug – саме для цього вона і призначена.

Налаштуйте середовище Visual Studio на Debug-конфігурацію. Потім потрібно встановити точки зупинок (*breakpoints англ.*). Точки зупинки ставляться у лівому стовпчику вікна вихідного тексту. Нехай нам потрібно відстежити роботу програми від початку - ставимо точку на першому рядку (рис. 18):

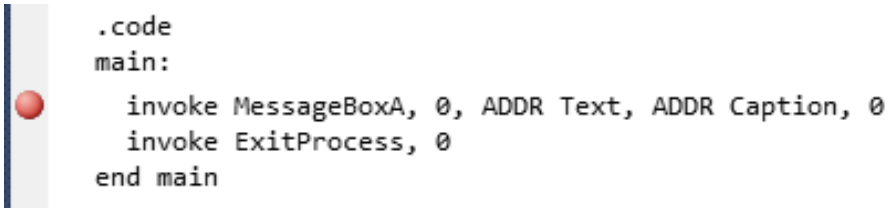


Рис. 18. Точка зупинки

Розпочинаємо налагодження, натиснувши клавішу F5. Програма не зупиняється, немов би не помічає точку зупинки. У чому справа? Порада: якщо точка зупинки на першій команді, то розпочинайте налагоджування, натиснувши клавішу F10.

Знову розпочнемо налагодження. Натиснемо клавішу F10. Тепер процес зупиняється у потрібній точці. Для перегляду значень регістрів відкрийте відповідне вікно через меню "Debug – Windows – Registers". Для аналізу створеного компілятором машинного коду відкрийте вікно дизасемблерного коду через меню "Debug – Windows – Disassembly".

У вікні "Disassembly" відображається машинний код, який згенерував компілятор. Це той код, який насправді виконує програма. Проте у вікні дизасемблера надається не власне машинний код (це множина двійкових чисел – кодів команд та даних), а текст у вигляді рядків на мові асемблеру як

результат **зворотного перетворення машинного коду у асемблер**. Дизасембльований текст найбільш близький до машинного коду – хіба що числові коди замінені символічними іменами команд та процедур. Тому цей текст дуже корисний для аналізу програми.

Не зважаючи на те, що ми пишемо вихідні тексти на асемблері, дизасембльований текст може суттєво відрізнятися від написаного програмістом вихідного тексту. Наприклад, замість **invoke** насправді буде декілька **push**, а потім **call**. Чим більше програміст використовує високорівневих синтаксичних конструкцій певної версії асемблера, тим більше вихідний текст програми відрізнятиметься від дизасембльованого (рис. 19).

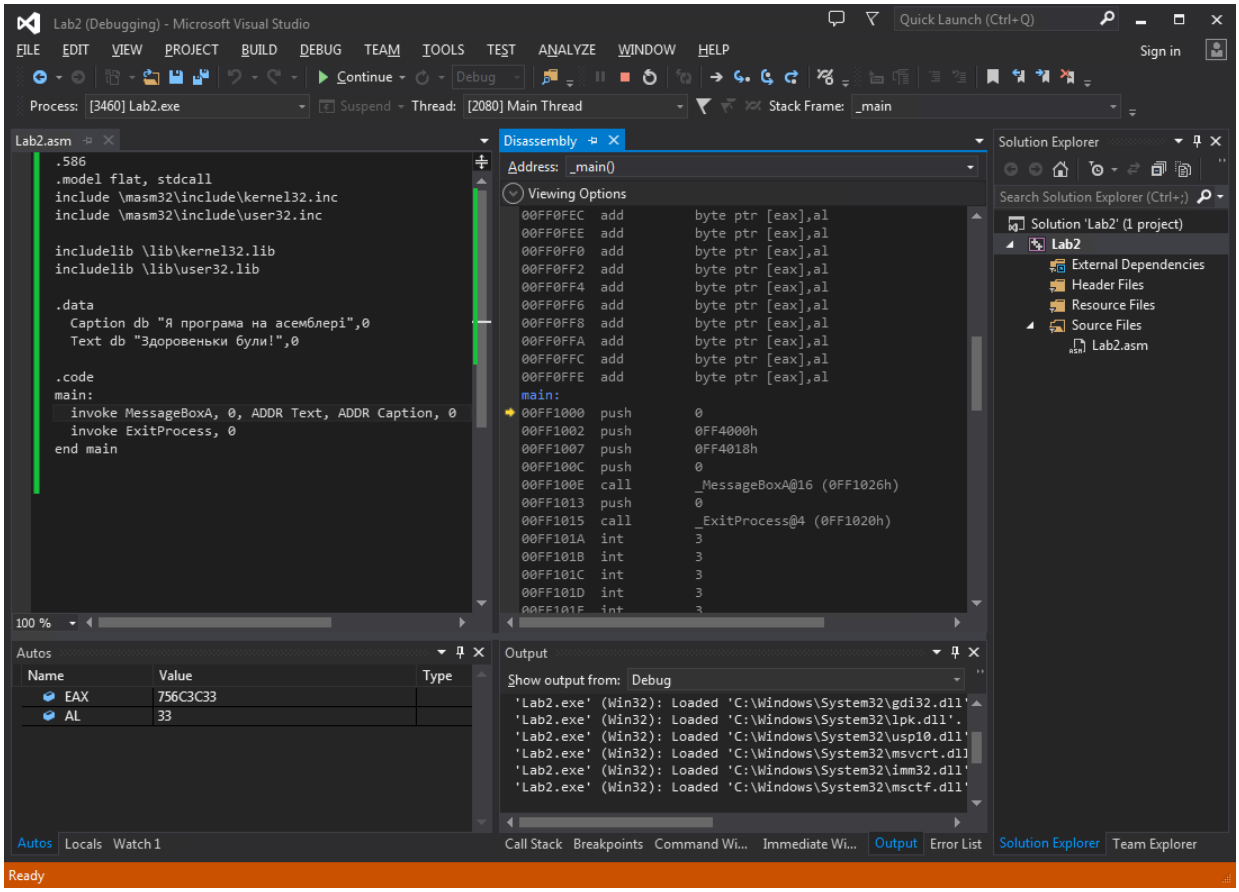


Рис. 19. Середовище Visual Studio у режимі стеження за покроковим виконанням програми

Для того, щоб простежити роботу програми по крокам натискуйте клавішу F10 або клавішу F11. Чим вони відрізняються? При натискуванні F11 (Step into) налагоджувач входить у процедури – так можна увійти у процедуру MessageBoxA і простежити її код та процес виконання. При натискуванні F10 (Step over) налагоджувач у середину процедур не входить і спостерігачу здається, що поточна процедура виконується немов би за один крок.

Варіанти завдання

Усім студентам необхідно запрограмувати:

- початкове діалогове вікно-вітання від автора програми;
- виконання команди CPUID з параметрами 0, 1, 2 а також 80000000h, 80000001h, 80000002h, 80000003h, 80000004h, 80000005h та 80000008h. Кожний результат виконання CPUID команди потрібно виводити у окремому діалоговому вікні. Якщо результати CPUID утворюють текстові дані, то виводити їх як рядки тексту.

Отримати дизасембльований код і проаналізувати його.

Пояснити значення N-го біту кожного результату команди CPUID, де N – номер студента у списку у журналі. Для пояснення використати документ "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference", доступний на сайті фірми Intel. Треба скачати цей документ.

Зміст звіту:

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми, основних результатів. Примітка. Задля дотримання всілякої конфіденційності у звіті не треба наводити роздруківку усіх кодів результатів, отриманих командами CPUID. Достатньо продемонструвати виконання програми на комп'ютері.
4. Аналіз, коментар вихідного тексту та дизасембльованого коду
5. Висновки

Контрольні питання:

1. Як розпочати проект на асемблері у середовищі MS Visual Studio?
2. Як додати файл вихідного тексту на асемблері у проект MS Visual Studio?
3. Як отримати виконуємий файл програми у середовищі MS Visual Studio?
4. Як налаштовувати конфігурації Debug та Release?
5. Як простежити роботу програми у налагоджувачі MS Visual Studio?
6. Що таке дизасембльований код?
7. Як можна проаналізувати машинний код?
8. Чим відрізняється **dword ptr** від **byte ptr** і навіщо вони потрібні?
9. Як задати параметр команді CPUID?